Electronic Notes in Theoretical Computer Science **20** (1999)
URL: http://www.elsevier.nl/locate/entcs/volume20.html 14 pages

Denotational Semantics of Shape: Past, Present and Future

C.B. Jay

School of Computing Sciences, University of Technology, Sydney
P.O. Box 123, Broadway NSW 2007, Australia;
Email: cbj@socs.uts.edu.au

Abstract

Past work on the semantics of vectors and arrays provides a denotational semantics for the new, higher-order, polymorphic array programming language FISh, that uses static analysis to determine array shapes. This semantics will be combined with that of shape polymorphism to underpin a language that will support both shape analysis and shape polymorphism on both arrays and inductive types.

Key words: shape, denotational semantics, FISh

1 Introduction

What is the denotational semantics of arrays? It is common to regard them as lists, or perhaps as (partial) functions on indices, but the need for a finer representation is clear from the study of matrices, which are not merely lists of lists. One way to restrict the latter is introduce sets of equational constraints on list lengths. Another approach uses the dependent type $\Sigma_{m,n:N} A^{m*n}$ to represent matrices with entries of type A but this requires either fixing the values of m and n or dynamic type-checking.

Both approaches have been adopted in practice, in traditions going back to Fortran and APL. Fortran began with a conservative view of array descriptions which was later liberalised by the use of compile-time constants, as in Pascal. However, this process never reached the flexibility of APL and its descendents, such as J [5] and MATLAB [22]. They are able to apply programs to arrays of arbitrary rank (number of dimensions) and size, but at the price of making type-checking a run-time option. The need for clear semantics and representations becomes even more pressing when using parallel languages, such as High Performance Fortran [24] or ZPL [25].

A tighter question is: how to distinguish one-dimensional arrays from lists? Here, even dependent types don't seem to help, as $\Sigma_{n:N} A^n$ is just the type of lists of A. In imperative languages like C, the distinction is typically based on

the time required to access entries; constant time versus linear. In functional languages like SML the distinction emphasises mutability; arrays are necessarily of reference type while typical lists are not. These answers are orthogonal to one another and neither of them distinguishes matrices from lists of lists.

So, despite being the first structured data types to enter general use, the typing and semantics of arrays are still problematic for the standard approaches.

In 1992 [7] I produced a denotational semantics for vectors and arrays, as follows. A *vector* is a list whose entries all have the same *shape*. A key point is that the shape of the vector is then a pair, consisting of its length and the common shape of the entries. Hence the entries in a vector of vectors all have the same length, making it a matrix. Of course, this approach is more useful if one has a general theory of shapes, and can use the theory to control array descriptions in an actual programming language. Unfortunately, the account of arrays was deemed too specialised, and without a language to model the work remained unpublished.

Now there is a general theory of shape [9] on which programming languages can be based. FISh [17,11] is a higher-order array programming language whose type system is strong, parametrically polymorphic and statically checked. It has a formal operational semantics [10] and a publically available implementation [4]. It can be viewed as the first Algol-like language [23,21] to support structured data types, and hence parametric polymorphism. Its array descriptions have been designed to support the maximum expressiveness consistent with static determination of array shapes. Its array type constructor represents arbitrary finite-dimensional regular arrays, i.e. hypercubes whose entries all have the same shape. That these need not be mutable improves support for referential transparency. Also, polymorphic programs, such as that for mapping, are able to act on array expressions of arbitrary dimension and size. Such power is nothing new to APL, but is achieved while maintaining static typing, and also static shape analysis.

Static *shape analysis* has many benefits, including increased polymorphism and static error-detection, and improved code optimisation. For example, it is used to convert mapping on a multi-dimensional array into a nested for-loop, to check that two arrays have equal length (to avoid array bound errors) or to support re-use of memory when two variables have the same shape. It also eliminates the need for boxing data, an overhead usually associated with polymorphism. For example, polymorphic quicksort applied to a large array of floats is twice as fast in FISh as in C [13].

Although FISh was based on novel categorical constructions, it does not yet have a formal denotational semantics. The first result of this paper is to model the FISh types using the semantics for arrays outlined above.

Another application of shape theory is *shape polymorphism* which allows programs to be polymorphic in their shape as well as in their data. The original ideas appeared in [16] and an experimental language P2 appeared shortly

after [8]. The conceptual framework was much improved by Moggi's idea of treating shape polymorphism as quantification over (categorical) functors. This led to the creation of Functorial ML (or FML) [14] which supports shape polymorphism over inductive data types.

Functors in FML can be modelled as categorical functors F which are *shapely over lists* [9]. In the single-sorted setting this amounts to having a cartesian natural transformation $F \Rightarrow L$ where L is the list functor. Intuitively this transformation extracts the data from the data structure and stores it in a list. It is easy to see that such functors are closed under elementary properties. The main theorem of [9] was to show that functors shapely over lists are closed under the construction of initial algebras.

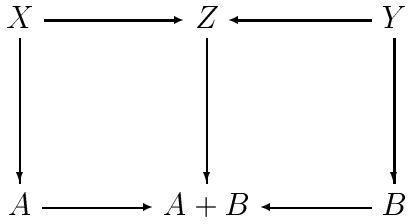
FISh 2 will support both shape polymorphism and shape analysis on both inductive types and arrays. It will do this by introducing a kind of FML functors extended with the vector functor V and some explicit shape analysis. As V is not shapely over L, the latter cannot act as the universal structure for data storage. Instead, the more general composite functor LV must be used. This semantics suggests that shape analysis will support run-time data storage in lists of arrays, reflecting both the irregular and regular aspects of data structures, with their respective flexibility and efficiency. The second result of this paper is to generalise the previous theorem to this new setting.

The body of the paper reviews the semantics of vectors and arrays, uses this to interpret the types of FISh 1, looks at the functors of FML and their semantics, and then at candidates for functors of FISh 2 and their denotational semantics. Conclusions and future work complete the paper.

2 Vectors

A *vector* is a list whose entries all have the same shape. To give this a categorical semantics we need to first fix our assumptions about the underlying category, and then provide a mechanism for describing shapes. Various alternatives could be considered, e.g. a fibrational setting, but it suffices to treat shaping as a morphism from an object A representing a data type to an object I representing the corresponding shapes. This can be formalised in the following categorical setting.

A *locos* $\mathcal{D}$ [3] is a lextensive category which has list objects. Lextensivity means that $\mathcal{D}$ has all finite sums and limits, and for any commuting diagram of the form



where the lower edge describes a coproduct, the upper edge describes a coprod-

uct iff the two squares are both pullbacks. In particular, products distribute over sums. The common definition of list objects is adequate if the locos is cartesian closed, but it must be parametrised in the general case. A locos can model booleans by $2 = 1 + 1$ and natural numbers by $N = L1$ where L is the list functor and 1 is the terminal object. Examples of locoses include the categories of sets and functions, bottomless c.p.o.'s and continuous functions, some categories of partial equivalence relations (PER's), toposes that have a natural numbers object, etc. Fix a locos $\mathcal{D}$ for the rest of the paper.

For any other category $\mathcal{I}$ the category $\mathcal{D}^{\mathcal{I}}$ of functors from $\mathcal{I}$ to $\mathcal{D}$ and natural transformations between them is also a locos, with all structure inherited pointwise from $\mathcal{D}$. Taking $\mathcal{I}$ to be the category $\cdot \to \cdot$ which has two objects and one non-trivial arrow between them yields the *arrow category* $\mathcal{D}^{\to}$ of $\mathcal{D}$. Its morphisms from the object $a : A \to I$ to the object $b : B \to J$ are pairs of arrows $f : A \to B$ and $u : I \to J$ making the following diagram commute:

(1)

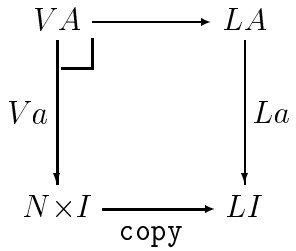
$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ I & \xrightarrow[u]{} & J. \end{array}$$

Now consider A as representing a type, and a as computing the shapes of its values, e.g. the shape of a matrix would determine the number of rows and columns. Then f can represent a function between types, whose action on shapes is given by u. That is, the shape $b\ (f\ x)$ is given by $u\ (a\ x)$. Using $\#$ to represent shape in all its forms we could then rewrite the diagram above as

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle \#}\downarrow & & \downarrow{\scriptstyle \#} \\ \#A & \xrightarrow[\#f]{} & \#B. \end{array}$$

As noted above, the list functor is given pointwise so that the list object for a above is $La : LA \to LI$. The *vector functor* $V : \mathcal{D}^{\to} \to \mathcal{D}^{\to}$ is derived from the list functor as follows. A vector is a list whose image under La is a list (of shapes) whose entries are all the same. Such lists are the image of the function $\mathtt{copy} : N \times I \to LI$ which maps a pair (n, i) to the list of length n whose entries are all i. Hence, vectors are obtained by pulling back the list

functor along copy, as follows:

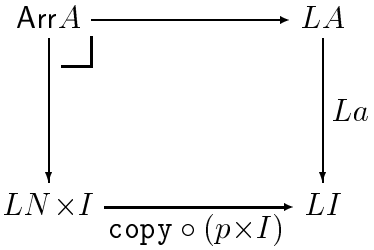


Note that the shape Va of a vector is given by its length and the common shape of all the entries.

Atomic data types, such as integers, are characterised by the property that all their values have the same shape, so we represent them by an arrow $D \to 1$ with trivial shape object. Then VD is isomorphic to LD and the vector/list distinction disappears. However, VD has non-trivial shape type that is isomorphic to N so that VVD (whose shape object is isomorphic to $N \times N$) can be used to represent matrices rather than arbitrary lists of lists.

This approach supports the usual matrix algebra quite comfortably. For example, transposition is a *total* operation on VVD (and is in fact a distributive law from the vector monad to itself). It supports sufficient reasoning about arrays to prove correctness of the fast Fourier transform [7].

Clearly, $VVVA$ represents three-dimensional regular arrays, etc. The shape of a k-dimensional array, i.e. an array of *rank* k, is given by a k-tuple of numbers, or sizes, paired with the common shape of all the entries. Indeed, we can construct a functor Arr to represent regular arrays of arbitrary rank, by using shapes that are lists of sizes paired with a common shape, as in



where p computes the product of the list of numbers.

This functor will be used to model the array type constructor of FISh 1. As the language also supports function types, it is well to consider those here, too.

Lemma 2.1 *If $\mathcal{D}$ is cartesian closed then so is $\mathcal{D}^{\to}$.*

Proof. Those pairs (f, u) as in (1) above for which $u \circ a = b \circ f$ can be

represented by c in the following pullback:

$$\begin{array}{ccc} E & \longrightarrow & A \to B \\ \downarrow{\scriptstyle c} & \lrcorner & \downarrow{\scriptstyle A \to b} \\ I \to J & \xrightarrow[a \to J]{} & A \to J \end{array}$$

The universal property follows directly. The result is a special case of Artin glueing with respect to the identity functor on $\mathcal{D}$ (see, e.g. [2]). □

Pullbacks are often used to introduce equational constraints on pairs. In the proof above the equations are in terms of functions $A \to J$. Equivalently, they can be regarded as equations on "elements" of J quantified over "elements" of A. In other words, the equations are quantified over A. This use of universal quantification is a little heavy-handed for the semantics of a type system. For example, it suggests that we could give a semantics to vectors of functions. However, checking that a list of functions is a vector would entail establishing extensional equality for their shapes. As these are also functions, it is impractical. On the other hand, if the pullbacks are restricted to those over an object with decidable equality then the lemma above does not hold. In any event, it seems wise to keep array and function types separate, as in FISh.

3 FISh 1

FISh 1 has been described in detail elsewhere [17,11,10,13]. Here we will focus on its types and their semantics. Its raw types are: The key division, in the

$$\begin{aligned} \delta : \mathsf{D} &::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{float} \mid \Gamma \mid \ \ldots \\ \alpha : \mathsf{A} &::= X \mid \delta \mid [\alpha] \\ \sigma : \mathsf{Sh} &::= \tilde{}\delta \mid \#\alpha \\ \tau : \mathsf{T} &::= \alpha \mid \sigma \\ \theta : \mathsf{P} &::= U \mid \#U \mid \mathsf{comm} \mid \mathsf{var}\ \alpha \mid \mathsf{exp}\ \tau \mid \theta \to \theta \\ \phi : \mathsf{S} &::= \theta \mid \forall X : \mathsf{A}.\ \phi \mid \forall U : \mathsf{P}.\ \phi. \end{aligned}$$

fi

style of Algol-like languages [23,21] is between *data types* τ which represent storable values, and *phrase types* θ, which represent meaningful program fragments. Data types split into *array types* α and *shape types* σ. The former consist of type variables, *datum types* δ used to represent atomic data, such as

integers or floats, and array types $[\alpha]$ used to represent regular arrays of finite rank. Shape types consist of static versions of the datum types and types of array shapes $\#\alpha$ for each array type α. All values of shape type are computed statically, during *shape analysis*, a form of partial evaluation [19] or abstract interpretation [1].

Phrase types are generated by phrase type variables and their shapes, a type comm of commands, a type of *array variables* var α for each array type α, a type of *expressions* exp τ for each data type, and are closed under *function types*. Commands are used to represent assignments, for- and while-loops, etc. var α represents storage for a value of type α. exp τ represents expressions that denote storable values (or shapes). Function types are as usual.

Earlier Algol-like languages did not support structured data types but used phrase types to represent, say, arrays. FISh has structured data types, and array type variables, so that it is meaningful to construct data polymorphic programs and use *type schemes* ϕ to represent them. Quantification over phrase type variables is also supported.

There is insufficient space here to treat the terms of the language, their shape analysis and evaluation, but a couple of examples may help to indicate their flavour. Assignment is given by a combinator $assign$: var $\alpha \to$ exp $\alpha \to$ comm. When the shape combinator # is applied to it the result reduces to

$$\lambda x, y.check\ (\#x\ \#=\ \#y)\ true$$

That is, whenever an assignment $x := t$ arises, shape analysis will check that x and t have the same shape, and report an error otherwise.

The language supports for- and while-loops in addition to the usual functional constructs. The standard prelude defines a higher-order function for mapping

$$map : \forall\alpha, \beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$$

When applied to a function and array it statically reduces to a nested for-loop. For example, consider the array expression x given by

```
new #y = {3:int_shape} in
  y[0] := 0; y[1]:= 1; y[2]:= 2
return y
```

It creates a local variable y whose shape is {3: int_shape} i.e. y is a vector of length 3 whose entries are of integer shape. Then $map\ (\lambda n.n+1)\ x$ reduces to a for-loop

```
new #z = {3:int_shape}
and #y = {3:int_shape} in
  y[0] := 0; y[1]:= 1; y[2]:= 2;
  for 0<= i<3 do z[i] := y[i]+1 done
return z
```

Now let us consider the semantics of the types with respect to a cartesian closed locos $\mathcal{D}$ that has countable sums, indicated by Σ. We can interpret the

closed data types as objects of $\mathcal{D}^{\rightarrow}$ (identifying the datum types with their standard representations):

$$\begin{aligned}
[\![\delta]\!] &= \delta + 1 \to 1 \\
[\![[\alpha]]\!] &= \mathsf{Arr}\ [\![\alpha]\!] \\
[\![\tilde{}\delta]\!] &= \mathtt{id} : \delta \to \delta \\
[\![\#\alpha]\!] &= \mathtt{id} : \#[\![\alpha]\!] \to \#[\![\alpha]\!].
\end{aligned}$$

The use of $\delta + 1$ to represent the datum type δ is to allow for the possibility of a datum whose value is unknown, as occurs when a data structure of known shape is created whose values are not yet initialised.

For the phrase types, we will interpret commands as store transformers, where a *store* is a function from locations to values. In more detail, let Loc be an object in $\mathcal{D}$ of *locations*. It is interpreted in $\mathcal{D}^{\rightarrow}$ as $\mathtt{id} : Loc \to Loc$. Define Val to be the sum $\Sigma_\alpha\ \mathsf{Arr}[\![\alpha]\!]$ in the arrow category, where the sum is over all array types α. Then the type of stores is $S = Loc \to Val$. The semantics of phrase types is given in the standard way by

$$\begin{aligned}
[\![\mathsf{var}\ \alpha]\!] &= (S \to [\![\alpha]\!]) \times ([\![\alpha]\!] \to S \to S) \\
[\![\mathsf{exp}\ \tau]\!] &= S \to [\![\tau]\!] \times S \\
[\![\mathsf{comm}]\!] &= S \to S \\
[\![\theta \to \theta']\!] &= [\![\theta]\!] \to [\![\theta']\!]
\end{aligned}$$

That is variables have values and can be assigned, expressions may have side-effects, commands are store transformers and functions are functions.

Of course, this only addresses the monomorphic setting, but there are no obvious barriers to extension to the polymorphic setting. FISh also employs local variables, whose modelling will require the sort of techniques introduced in [21] (especially Volume 2).

4 Functorial ML

Functorial ML (or FML) [14] is an extension of the core of ML with a type class of functors for inductive data types (not to be confused with the functors on signatures of Standard ML). Then shape polymorphism is expressed using quantification over functors. The raw syntax for the FML functors is given by

$$F, G ::= X \mid C \mid \Pi_i^m \mid F\langle \overline{G} \rangle^n \mid \mu^m F$$

Every functor F has an *arity* m written $F : m$ which indicates the number of its arguments. X is a functor variable, C represents constants, such as the product $\times$ and sum $+$. The *projection functor* Π_i^m picks out the ith argument of m. The composite of F with the tuple of functors $\overline{G} = G_0, G_1, \ldots, G_{m-1}$ is $F\langle \overline{G} \rangle^m$ and $\mu^m F$ is the initial algebra functor for F with respect to its last argument.

FML supports primitives for mapping, folding and zipping at each arity that are polymorphic in the choice of functor. For example, we have

$$map_m : \forall F : m.\forall \overline{X}, \overline{Y}.\ \overline{X \to Y} \to F\overline{X} \to F\overline{Y}$$

This differs from Jones constructor classes [20] in that the combinators all act parametrically without the need for dictionaries. It is rather closer to work on *polytypy* [18,6] but is able to handle all arities of functor instead of just $n = 1$ or $n = 2$ [15]. The FML functors can be modelled categorically using ideas from [9] which we will now review.

A *strength* for a functor $F : \mathcal{D}^m \to \mathcal{D}$ is a natural transformation $\tau_{A,B} : FA \times B \to F(A \times B)$ which satisfies the standard unicity and associativity conditions with respect to $\times$. The definition extends pointwise to functors $F : \mathcal{D}^m \to \mathcal{D}^n$. Such an F is *shapely* if it preserves all pullbacks. The list functor is our canonical example of a shapely functor. More generally, we will favour the functor $\Pi L : \mathcal{D}^m \to \mathcal{D}$ where

$$\Pi L(A_0, \ldots, A_{m-1}) = LA_0 \times \cdots \times LA_{m-1}$$

as this will allow us to handle several kinds of data simultaneously.

A natural transformation $\delta : F \Rightarrow G$ is *cartesian* if each of its defining squares

$$\begin{array}{ccc} FA & \xrightarrow{\delta_A} & GA \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ FB & \xrightarrow[\delta_B]{} & GB \end{array}$$

is a pullback. If, in addition, F and G are shapely functors and δ commutes with the strengths (in the obvious way) then δ is a *shapely transformation* and we say that F is *shapely over* G. When G is the shapely functor ΠL then F is a *shapely type constructor*. These concepts can be generalised to functors $\mathcal{D}^m \to \mathcal{D}^n$ by requiring the property for all projections to $\mathcal{D}$.

The product functor is always shapely over lists, and the sum is so because of its stability under pullback. Also, such functors are closed under composition because the multiplication of the list monad $flat : LL \Rightarrow L$ (also known as flattening) is cartesian. That is, if $\delta_i : F_i \Rightarrow L$ are cartesian natural transformations then so is

$$F_1F_0A \xrightarrow{\delta_0\delta_1} LLA \xrightarrow{flat} LA$$

Thus, to model the FML functors by shapely type constructors it suffices to show that the latter are closed under initial algebras. Given $F : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$ let $F^\dagger : \mathcal{A} \to \mathcal{B}$ denote its initial algebra functor (if it exists). Also, given a natural transformation $\beta : F\langle \mathtt{id}, G\rangle \Rightarrow G$ then $\beta^\dagger : F^\dagger \Rightarrow G$ is the induced natural transformation.

Theorem 4.1 *If $F : \mathcal{D}^{m+n} \to \mathcal{D}^n$ is a shapely type constructor then $F^\dagger : \mathcal{D}^m \to \mathcal{D}^n$ exists and is one, too. Further, if $\beta : F\langle \mathtt{id}, G\rangle \Rightarrow G : \mathcal{D}^m \to \mathcal{D}^n$ is a shapely transformation, then so is $\beta^\dagger : F^\dagger \Rightarrow G$.*

Proof. See [9, Section 8]. The fundamental idea is that initial algebras can be recognised by a parser, which can be built using the list technology of a locos. □

5 FISh 2

FISh version 2 will combine the benefits of FISh with those of FML. That is, it will support both array types and inductive types, shape polymorphism and shape analysis. Shape analysis will support improved execution of shape polymorphic programs, much as occurs in FISh 1. Execution speed is anticipated to maintain the relativities established by FISh 1, i.e. significantly faster than other higher-order polymorphic functional programming languages, and comparable to C.

Here is the core of the type system.

$$F : \mathsf{F} ::= X \mid C \mid F, F \mid L \mid R \mid F\ F \mid \mu\ F$$

$$\theta : \mathsf{P} ::= U \mid \mathsf{var}\ F \mid \mathsf{exp}\ F \mid \theta \to \theta$$

$$\phi : \mathsf{S} ::= \theta \mid \forall X : \mathsf{F}.\ \phi \mid \forall U : \mathsf{P}.\ \phi$$

It is much simpler than either of the type systems of FISh 1 or FML. First, the idea of separate types for shapes has been dropped. Now a term and its shape have the same type. This increases the expressive power of the language (at the cost of some complexity in reasoning about programs). Second, functor arities are no longer explicit. This is achieved by allowing functors to produce multiple arguments, corresponding to categorical functors $\mathcal{D}^m \to \mathcal{D}^n$ where n may now be greater than 1. This allows us to decompose functor composition $F\langle G, H\rangle^n$ into functor pairing $K = (G, H)$ and a simple form of composition $F\ K$. Projections Π_i^m are replaced by composites of the projections L and R for the functor pairing. The absence of explicit arities means that indices can be dropped from the language constants, a major simplification.

The functor constants C will now include the vector functor V as well as the usual binary products and sums. This adds significantly to the expressive power of the language and to the demands placed upon the semantics. We may write $X * Y$ for $product(X, Y)$ or $X + Y$ for $coproduct(X, Y)$.

FISh 2 will support a function `map` of type

$$map : \forall F, X, Y.\ (X \to Y) \to (FX \to FY)$$

Note that X and Y may be given by pairing of functors, so that all of the expressive power of mapping in FML is retained.

Examples of the new data types available include the quad-trees and oct-trees of arrays used to divide up two- and three-dimensional space. More

generally, they include types which support an irregular outer structure, e.g. some kind of tree, and a regular inner structure, e.g. at the leaves. Such data types can be implemented using pointers, e.g. a linked list, to blocks of memory. FISh 2 will convert key operations, such as mapping and reducing, into pattern-matching over the tree structure followed by for-loops over array entries.

When vector types interact with inductive types then some interesting phenomena arise. It is well known how to decompose a list of pairs into a pair of lists $L(A \times B) \to LA \times LB$ and that this is reversible if the lists have the same length. The same can be done with vectors. For sums the situation is different. $L(A + B)$ does not support any such decomposition but

$$V(A + B) \cong VA + VB$$

since the choice of being an A or a B is part of the shape, and so is common to all entries. Similarly, the length of a list is part of its shape, so that $V(LA) \cong V(VA)$. The general pattern is shown by any tree functor T for which we have

$$V(TA) \cong T'(VA)$$

where the entries in T' must all have the same shape. As all the entries in a vector of trees must have the same shape, it follows that we can 'transpose' the structure (in the sense of [7]) to obtain a single tree whose labels are arrays, i.e. to move all the irregular structure to the outside. This result suggests significant benefits in implementation, the former type requires a vector of tree structures, whereas the latter requires only one, labelled by vectors. A thoroughgoing approach to optimising data representation will require further semantic analysis of this class of isomorphisms.

A semantics for FISh 2 must unify the semantics of its two predecessors. At first glance, this may appear to be simple. After all, each array type constructor, e.g. the matrix functor $M : \mathcal{D} \to \mathcal{D}$, is shapely over lists and so the framework developed for FML would appear to suffice. However, this functor on $\mathcal{D}$ only ensures that the matrix is rectangular, without placing any constraint on the shapes of the entries. For this we must move to the arrow category, and use V.

Now V is not shapely over the list functor on $\mathcal{D}^{\to}$ as the canonical natural transformation from V to L is not cartesian (the component at $1 \to 1$ is an identity morphism) so we must consider a more general notion of data than a mere list. Consider a type of trees of arrays as above. The tree structure can be flattened to a list, and the array structure to a vector, to get a cartesian natural transformation to LV. At first glance, this asymmetrical functor may not appear intuitive but it allows us to represent data structures with an irregular outer structure with islands of regularity within.

Once this viewpoint has been adopted, the results for functors shapely over lists on $\mathcal{D}$ can be generalised to functors shapely over LV on $\mathcal{D}^{\to}$. For the purposes of this paper alone, let us call these the *shapely array type construc-*

tors. As both L and V are shapely over LV both the vector functor and all of the functors shapely over lists are shapely array type constructors. Such functors are closed under composition because LV has a cartesian flattening transformation $LVLV \Rightarrow LV$ just like L. Finally, there is the question of initial algebras.

Theorem 5.1 *If* $F : (\mathcal{D}^{\rightarrow})^{m+n} \rightarrow (\mathcal{D}^{\rightarrow})^{n}$ *is a shapely array type constructor then* $F^{\dagger} : (\mathcal{D}^{\rightarrow})^{m} \rightarrow (\mathcal{D}^{\rightarrow})^{n}$ *exists and is one, too. Further, if* $\beta : F\langle \mathit{id}, G\rangle \Rightarrow G : (\mathcal{D}^{\rightarrow})^{m} \rightarrow (\mathcal{D}^{\rightarrow})^{n}$ *is a shapely transformation, then so is* $\beta^{\dagger} : F^{\dagger} \Rightarrow G$.

Proof. Adapt the proof of Theorem 4.1. The only change required is to modify the parsing recognition test in [9, Lemma 8.1] to check the regularity condition necessary for vectors. □

Thus, the shapely array type constructors will be used to model all of the functors of FISh 2. Phrase types will be handled just as in FISh 1.

6 Conclusions

Shape theory uses insights from denotational semantics to construct a new interpretation of array types and, more generally, a new approach to computing with data structures. Concerning arrays, it replaces the operational characterisation of arrays in terms of their access time or mutability by a denotational requirement that their entries all have the same shape. FISh 1 shows that this characterisation can be exploited during static program analysis to support static shape checking and program optimisation. FISh 2 will combine the benefits of shape analysis with those of shape polymorphism, as developed in FML. Finally, note that knowledge of shapes is of critical importance for parallel programming. A parallel version of FISh called **GoldFISh** is also under development [12].

Acknowledgement

The anonymous referees, Gabi Keller and Manuel Chakravarty made many useful comments and suggestions.

References

[1] S. Abramsky and C. L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Computers and Their Applications, Ellis Horwood, 1987.

[2] A. Carboni and P. T. Johnstone, *Connected limits, familial representability and artin glueing,* Mathematical Structures in Computer Science **5** (1995), pp. 441–459.

[3] J. R. B. Cockett, List-arithmetic distributive categories: locoi, Journal of Pure and Applied Algebra **66** (1990), pp. 1–29.

[4] Fish web-site, URL: `http://www-staff.socs.uts.edu.au/~cbj/FISh`.

[5] K. E. Iverson, "J Introduction and Dictionary." Iverson Software Inc. (ISI), 1995.

[6] P. Jansson and J. Jeuring, *PolyP - a polytypic programming language extension,* In: *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* ACM Press, 1997, pages 470–482.

[7] C. B. Jay. *Matrices, monads and the fast Fourier transform,* In: *Proceedings of the Massey Functional Programming Workshop 1994*, 1994, pages 71–80.

[8] C. B. Jay, *Polynomial polymorphism,* In: R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference: Glenelg, South Australia 1–3 February, 1995*, A.C.S. Communications **17** (1995), pp. 237–243.

[9] C. B. Jay, *A semantics for shape,* Science of Computer Programming **25** (1995), pp. 251–283.

[10] C. B. Jay, *The FISh language definition,* URL: `http://www-staff.socs.uts.edu.au/~cbj/Publications/fishdef.ps.gz`, 1998.

[11] C. B. Jay, *Poly-dimensional array programming,* URL: `http://www-staff.socs.uts.edu.au/~cbj/Publications/polydimensional2.ps.gz`, August 1998.

[12] C. B. Jay, *Costing parallel programs as a function of shapes,* Science of Computer Programming, 1999, in press.

[13] C. B. Jay, *Partial evaluation of shaped programs: experience with FISh,* In: O. Danvey, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99) San Antonio, Texas, January 22-23, 1999: Proceedings* BRICS, 1999, pages 147–158.

[14] C. B. Jay, G. Bellè, and E. Moggi, *Functorial ML,* Journal of Functional Programming **8** (1998), pp. 573–619.

[15] C. B. Jay, E. Moggi, and G. Bellè, *Functors, types and shapes,* In: R. Backhouse and T. Sheard, editors, *Workshop on Generic Programming: Marstrand, Sweden, 18th June, 1998*, Chalmers University of Technology, 1998, pages 21–4.

[16] C. B. Jay and J. R. B. Cockett, *Shapely types and shape polymorphism,* In: D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, Lecture Notes in Computer Science, Springer Verlag, 1994, pages 302–316.

[17] C. B. Jay and P. A. Steckler, *The functional imperative: shape!,* In: Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming,* ESOP'98. Held as part of the joint european conferences on theory and practice of software, ETAPS'98, Lisbon, Portugal, March/April 1998, Lecture Notes in Computer Science **1381** (1998), pp. 139–53.

[18] J. Jeuring, *Polytypic pattern matching,* In: *Conference on Functional Programming Languages and Computer Architecture*, 1995, pages 238–248.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft, "Partial Evaluation and Automatic Program Generation," International Series in Computer Science, Prentice Hall International, 1993.

[20] M. P. Jones, *A system of constructor classes: overloading and implicit higher-order polymorphism,* J. of Functional Programming **5** (1995).

[21] P. W. O'Hearn and R. D. Tennent, editors, "Algol-like Languages, Vols I and II," Progress in Theoretical Computer Science, Birkhauser, 1997.

[22] D. Redfern and C. Campbell, "The MATLAB 5 Handbook," Springer-Verlag, 1998.

[23] J. C. Reynolds, *The essence of ALGOL,* In: J. W. de Bakker and J. C. van Vliet, editors, "Algorithmic Languages," IFIP, North-Holland Publishing Company, 1981, pp. 345–372.

[24] H. Richardson, *High Performance Fortran: history, overview and current developments,* URL: `http://www.crpc.rice.edu/HPFF/publications.html`, 1996.

[25] University of Washington. ZPL, URL: `http://www.cs.washington.edu/research/zpl/index.html`, 1997.